Sistema de Monitoreo y Control de Invernaderos Inteligentes (SIMCII)

Documentación técnica

Escuela:

Instituto Tecnológico de Orizaba

Materia:

Tecnologías de Programación

Docente:
Dr. Ulises Juárez Martínez

Integrantes:

Jonathan Uriel Vega Román

Emmanuel Torres Martínez

Rubén Castillo

Contenido

Sistema de Monitoreo y Control de Invernaderos Inteligentes (SIMCII) 1
1) Documentación técnica 3
1.1 Resumen ejecutivo 3
1.2 Componentes y responsabilidades 3
1.3 Requisitos mínimos 3
1.4 Endpoints principales 3
1.5 Secuencia de arranque 4
1.6 Troubleshooting 4
2) Diagrama de clases 4
3) Manual técnico 5
3.1 Objetivo 5
3.2 Estructura del repositorio (resumen) 5
3.3 Variables de entorno críticas 5
3.4 Comandos frecuentes 5
3.5 Health checks y monitoreo 5
3.6 Desarrollo y pruebas locales 5
Documentación Web 5

1) Documentación técnica

1.1 Resumen ejecutivo

SIMCII es una plataforma microservicios para gestión de dispositivos IoT y alertas. Está compuesta por cinco contenedores orquestados con Docker Compose: PostgreSQL, Java Service (Spring Boot), Python Service (FastAPI), Go Gateway (autenticación + routing) y Frontend (Nginx que sirve la SPA). El gateway expone la entrada pública y maneja autenticación (Firebase / JWT). Los servicios backend exponen APIs REST y puntos de salud para verificación.

1.2 Componentes y responsabilidades

- **iot-postgres**: almacén relacional (PostgreSQL 15). Provee conexión para el Java Service. Se espera que pg_isready confirme salud.
- **iot-java-service**: Spring Boot que implementa CRUD de dispositivos y persistencia con Hibernate. Exposición de /api/... y /actuator/health.
- **iot-python-service**: FastAPI, responsable del sistema de alertas (/api/alertas/activas) y chequeos de salud /health.
- **iot-go-gateway**: Gateway en Go que administra autenticación (Firebase + JWT), enrutamiento hacia los microservicios y sirve login/dashboard en :8081.
- **iot-frontend**: Nginx sirviendo assets (puerto 3000 en modo directo), interfaz de usuario.

1.3 Requisitos mínimos

- Docker >= 20.10, Docker Compose >= 2.0, Git, curl.
- Recursos: 4 GB RAM mínimo, 10 GB disco.
- Puertos libres: 3000, 5432, 8000, 8080, 8081.
- Opcional: proyecto Firebase con Authentication y serviceAccountKey.json para operaciones de admin.

1.4 Endpoints principales

- Gateway (público): http://localhost:8081 (login /login, dashboard /dashboard).
- Frontend directo: http://localhost:3000.
- Java API: http://localhost:8080/api/dispositivos y http://localhost:8080/actuator/health.
- Python API: http://localhost:8000/api/alertas/activas y http://localhost:8000/health.

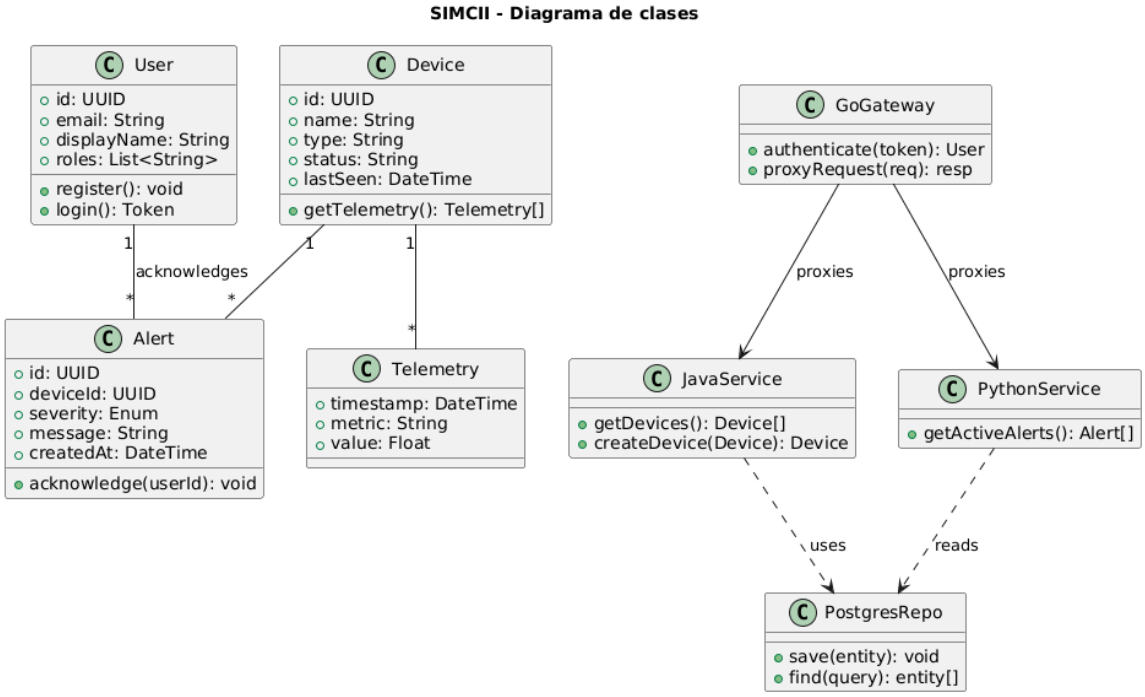
1.5 Secuencia de arranque

Orden esperado (Compose health checks): iot-postgres → iot-java-service → iot-frontend (arrancado después de postgres) → iot-python-service (depende java) → iot-go-gateway (depende java y frontend).

1.6 Troubleshooting

- Consultar docker compose logs -f y docker compose ps.
- Comprobar health endpoints: 200 / { "status":"UP" } según servicio.
- Si Java no crea tablas: revisar SPRING_JPA_HIBERNATE_DDL_AUTO.
- Problemas comunes: conflictos de puerto, credenciales de DB, ausencia de serviceAccountKey.json para Firebase Admin.

2) Diagrama de clases



Este diagrama se encuentra dentro de la carpeta docs del repositorio.

3) Manual técnico

3.1 Objetivo

Proveer procedimientos para desarrollar, desplegar y operar los microservicios SIMCII en un entorno local y para debugging.

3.2 Estructura del repositorio (resumen)

- docker-compose.yml — orquestación y health checks.
- go-gateway/ — código Go, Dockerfile, .env con API keys de Firebase.
- java-service/ — Spring Boot app con application.yml y Dockerfile.
- python-service/ — FastAPI app con dependencias y Dockerfile.
- frontend/ — assets y build pipeline servidos por Nginx.

3.3 Variables de entorno críticas

- POSTGRES_USER, POSTGRES_PASSWORD, POSTGRES_DB (iot-postgres)
- SPRING_DATASOURCE_URL, SPRING_DATASOURCE_USERNAME, SPRING_DATASOURCE_PASSWORD, SPRING_JPA_HIBERNATE_DDL_AUTO
- FIREBASE_API_KEY (Go Gateway)
- GO_GATEWAY_PORT=8081 (puede configurarse)

3.4 Comandos frecuentes

- Iniciar: docker compose up --build -d
- Logs: docker compose logs -f --tail=200
- Ver estado: docker compose ps
- Forzar rebuild de un servicio: docker compose up -d --build java-service
- Ejecutar migraciones (si aplica): revisar scripts en java-service (Hibernate auto ddl configurado).

3.5 Health checks y monitoreo

- Java: /actuator/health devuelve { "status":"UP" }.
- Python: /health devuelve 200.
- Postgres: pg_isready.
- Configurar Prometheus/Grafana (recomendado) para métricas; instrumentar endpoints si se requiere.

3.6 Desarrollo y pruebas locales

- Modificar código y reconstruir la imagen del servicio correspondiente.
- Para desarrollo iterativo en Java/Python, preferible levantar la app localmente (no en contenedores) con la DB en Docker o una instancia local.
- Uso de Postman/curl para probar endpoints.

Documentación Web

Si desea consultar la documentación con mayor detalle puede dar clic aquí.